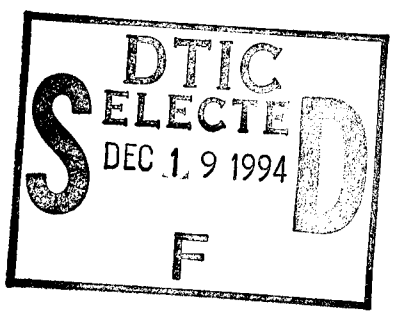
AEROSPACE REPORT NO.
ATR-92(2778)-5

The Semantics of Ada Access Types (Pointers) in the State Delta Verification System (SVDS)



30 September 1992

Prepared by

L. G. MARCUS
Computer Systems Division

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Engineering and Technology Group

THE AEROSPACE CORPORATION
El Segundo, California

Aerospace Report No.
ATR-92(2778)-5

THE SEMANTICS OF ADA ACCESS TYPES (POINTERS) IN THE STATE DELTA VERIFICATION SYSTEM (SDVS)

Prepared by

L. G. Marcus
Computer Systems Division

30 September 1992

Engineering and Technology Group
THE AEROSPACE CORPORATION
El Segundo, CA 90245-4691

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Report No.
ATR-92(2778)-5

THE SEMANTICS OF ADA ACCESS TYPES (POINTERS) IN
IN THE STATE DELTA VERIFICATION SYSTEM (SDVS)

Prepared

L. G. Marcus

Approved

B. H. Levy, Manager
Computer Assurance Section

D. B. Baker, Director
Trusted Computer Systems Department

C. A. Sunshine, Principal Director
Computer Science and Technology
Subdivision

Abstract

We propose a method for handling Ada access types (pointers) in SDVS, i.e., a method that allows SDVS to translate and reason about Ada programs containing access types. The method is built upon "higher-order places", i.e., "places" that have other places as contents.

We give the state delta semantics for various Ada constructs involving access types, discuss the theory and implementation of higher-order places, and give a partially worked example of an Ada program involving access types.

Contents

1 Ada Access Types in SDVS

We propose a method for handling Ada access types (pointers) in SDVS, i.e., a method that allows SDVS to translate and reason about Ada programs containing access types. As in any enhancement of SDVS to handle a language extension, we need to find the proper translation of the new construct into SDVS and then to add any necessary processing and proving capabilities involving the new construct.

In this case we are lucky. It appears that the semantics of Ada access types corresponds exactly to a pre-existing aspect of the theory of SDVS, namely places that have other places for their contents (or simply "places of places"), and only a small amount of enhancement is necessary in order for SDVS to be able to deal practically with the various ramifications of places of places. The fact that the state delta logic has an explicit function symbol ("dot") for the contents of a place, instead of considering contents as simply the interpretation of a variable via an "evaluation" as is customary in temporal logic, makes the theory of places of places very natural.

To quote from the Ada Reference Manual (LRM) [1] (Section 3.8, on access types):

> Access to such an object is achieved by an *access value* returned by an allocator; the access value is said to *designate* the object.

The value designates an object; that is exactly the situation of a place whose contents (value) is another place. This type hierarchy on places has been previously exploited in SDVS only in connection with new and old "universes" of places corresponding to entry and exit from block environments. The theory is discussed in [2].

In addition to [1], our thoughts and examples below are based on reading [3], [4], and [5].

In some operations on an access type variable x in Ada, the loose phrase "contents of x" could be interpreted as $.(.x)$ (when we are interested in the value of the object x points to, designated in Ada by *x.all*), and in other contexts, when we are interested in the thing itself that x points to, as $.x$ (designated in Ada simply by x). Even though $.x$ is a place, it is not a "predeclared" place; it does not have an independent declaration. It can only be referred to by $.x$. The contents of a pointer place consist of only one place, not perhaps many, as in the case of "universes." Thus, it makes sense to write $..x$ or $\#.x$. (In actual SDVS this must be written with parentheses as $.(.x)$ or $\#(.x)$.) An assignment statement of the form $a := b$, where a and b are access type objects, would be translated by a state delta with $\#a = .b$ in its postcondition and a modification list consisting of a. An assignment statement of the form $a.all := b.all$ could involve a state delta with modlist $.a$ and either $\#(.a) = .(.b)$ or $\#(\#a) = .(.b)$ in the postcondition; these two should be equivalent since (the place designated by) a does not change during that assignment, so $\#a$ is equal to $.a$ at postcondition time.

Note that Ada can have pointers of pointers. Thus we may need $.(.(.x))$, etc. [1]:

> There are no particular limitations on the designated type of an access type.

In particular, the type of a component of the designated type can be another access type, or even the same access type.

This raises some issues when a proof may require an indefinite number of iterations of the "dot" operation, or a record access type may require an arbitrary depth expression of the form $record(.record(.record \ldots f), f)$. (See Section 6.)

The pointers x and y are said to be equal ($x = y$) at a given time (in a given state) if $.x = .y$; thus it is necessary, but not sufficient, that $.(.x) = .(.y)$.

For example, the following program returns "maybe":

```
with text_io; use text_io;
procedure pointer1 is
     type text is access string;
     a:  text := new string'("monday");
     b:  text := new string'("monday");
begin
     if a = b then
     put("yes");
     elsif a.all = b.all then
     put("maybe");
     else put("nope");
     end if;
end pointer1;
```

A "null" pointer is one that is not pointing to anything, i.e., if x is a null pointer, then $.x$ simply does not have an explicit contents, i.e., $.x$ does not designate a specific place, and $.(.x)$ "does not exist." However, *null* is the "non-place" that such an access type object "non-designates," and all such places test as being equal in Ada. See the programs *pointer2, pointer3*, and *pointer4* below. This situation is brought about by an access type declaration with no **new** value assignment, or by an explicit assignment of the value *null.*

Note that the two programs below both return "yes":

```
with text_io; use text_io;
procedure pointer2 is
     type text is access string;
     a: text;
     b: text;
begin
     if a = null then
     put("yes");
     else put("no");
     end if;
end pointer2;
```

```
with text_io; use text_io;
procedure pointer3 is
     type text is access string;
     a: text;
     b: text;
begin
     if a = b then
     put("yes");
     else put("no");
     end if;
end pointer3;
```

The following program causes a constraint error because *a.all* "does not exist" for $a = null$.

```
with text_io; use text_io;
procedure pointer4 is
     type text is access string;
     a: text;
     b: text;
begin
     if a.all = b.all then
     put("yes");
     else put("no");
     end if;
end pointer4;
```

What is the difference between the fragment

```
procedure pointer is
     type access_char is access character;
     a: access_char;
begin
     a := new character;
```

and just plain

```
procedure pointer is
     type access_char is access character;
     a: access_char;
```

with no assignment? The difference is that in the second case $a = null$ (and so *a.all*, or .(*a*) in SDVS, "does not exist"), which is not true in the first case. The use of "new" in the Ada LRM is somewhat mysterious. The original Ada LRM does not really explain its meaning, and one edition of the annotated LRM [1] even has this cryptic note in the index:

> Index has no entry for "NEW," a reserved word pertaining to access variables.

A "constant" pointer x cannot have different values for $.x$. Of course, it can have different $.(.x)$ values.

> If an access object is constant, the contained access value cannot be changed and always designates the same object. On the other hand, the value of the designated object need not remain the same [1].

For example, note that the following program is illegal, because it assigns to a constant (the line $b := a$), even though the contents of the places designated by a and b are the same.

```
with text_io; use text_io;
procedure pointer5 is
     type text is access string;
     a: constant text := new string'("monday");
     b: constant text := new string'("monday");
begin
     b := a;
     put(b.all);
end pointer5;
```

Another verification system intended to handle Ada is Penelope, being developed at ORA Corporation. It is not clear from [6] whether or not Penelope handles access types. In an email exchange with David Guaspari [7], he clarified that the "predicate transformers associated with the declaration of access types, of access variables, of allocators, etc. are all defined, but none of them has been implemented in Penelope." This seems to be approximately equivalent to the shape SDVS is in with regard to access types.

Of course, many other researchers have examined formalizations of pointers, among them [8]-[18].

2 More Details on the Translation

There are essentially three kinds of Ada constructs that must be translated into SDVS: declarations of access types, assignments (to the access objects or objects they designate), and equality comparisons (between access objects or between objects they designate).

New features needed are

1. *incomplete* and circular type declaration
2. *access* type declaration
3. *null* place (does not have dot value)

Below are some examples of Ada pointer constructs. D1 and D2 are assumed to be previously declared access type variables.

- D1:P; – equivalent to D1:P:=null;
- if D1 = null then ....
- D2 := new DATE'(4, JUL, 1776); – creates and gives value.
- D1.day := 12;
- D1.all:= (12, OCT, 1492);
- D2.all := D1.all; – copying the record, i.e., #(.D2) = .(.D1)
- D2 := D1; – copying the pointer, i.e., #D2 = .D1; D2 now points to where D1 points, forcing all values to keep in sync from this point on (until another pointer copying assignment occurs; so following this statement by D3:=D2 has the same effect as following it with D3:=D1).
- Incomplete type declarations, in order to define access types that are in essence circular: e.g.

```
type LINK;
type P is access LINK;
type LINK is
   record
      INT: INTEGER;
      NEXT: P;
   end record;
```

With regard to the difference between copying the record and copying the pointer, note that in the following two programs that the first returns "z", while the second returns "v."

```
with text_io; use text_io;
procedure pointer6 is
     type access_char is access character;
     a, b: access_char;
begin
     a := new character'('z');
     b := new character;
     b.all := a.all;
     a.all := 'v';
     put(b.all);
end pointer6;

with text_io; use text_io;
procedure pointer7 is
     type access_char is access character;
     a, b: access_char;
begin
     a := new character'('z');
     b := new character;
     b := a;
     a.all := 'v';
     put(b.all);
end pointer7;
```

Note that it is necessary to include the two lines with "new" in each of the above procedures, in the sense that omitting either one will cause a constraint error exception when the compiled code is executed.

The proposed translation of *procedure pointer6* above would entail the application of the following state deltas, after the declaration phase:

```
[sd pre: true
  comod: (all)
    mod: (.a)
   post: (#(.a) = 'z' and s1)]
```

where s1 is

```
[sd pre: true
  comod: (all)
    mod: (.b)
   post: (s2)]
```

where s2 is

```
[sd pre: true
  comod: (all)
    mod: (.b)
   post: (#(.b) = .(.a) and s3)]
```

where s3 is

```
[sd pre: true
  comod: (all)
    mod: (.a)
   post: (#(.a) = 'v' and s4)]
```

where s4 is

```
[sd pre: true
  comod: (all)
    mod: (stdout)
   post: (#stdout = .(.b))]
```

whereas for *procedure pointer7* s2 is replaced by

```
[sd pre: true
  comod: (all)
    mod: (b)
   post: (#b = .a and s3)]
```

Note that in the following program the first assignment is illegal, since a and b are of different base types. But since they both designate objects of type *integer*, the second assignment is allowed.

```
procedure pointer8 is
     type T1 is access integer;
     type T2 is access integer;
```

```
     a : T1 := new integer'(5);
     b : T2 := new integer;
begin
--   b := a; -- illegal
     b.all := a.all;  --legal
end pointer8;
```

3 SDVS Processing of Places of Places

There are four possible combinations of dots and pounds for contents of places of places:

```
.(.x), #(#x), #(.x), .(#x)
```

In postconditions of state deltas, where simultaneous use is made of the old contents before the state change and the new contents after the state change, these would refer to, respectively, the old contents of the old place pointed to by x, the new contents of the new place pointed to by x, the new contents of the old place pointed to by x, and the old contents of the new place pointed to by x. SDVS 11 correctly (at least with respect to our interpretation of the semantics of places of places) handles occurrences of the first three, but not the fourth.

But that is almost "okay," because it does not appear that the latter expression is ever going to come up in the translation of an Ada program (of course it could come up in the specification). In other words,

```
.(#x)
```

cannot be expressed as some *x.all*; there is no Ada statement that refers to the *previous* contents of a current access object, unless of course that current access object is the same as the previous one designated by x, in which case

```
.(.x)
```

will suffice.

It is also obvious that our solution below to the problem of $.(\#x)$ would not help if that occurrence were due to an Ada program, since the object designated by an access type object is not allowed to be previously named. (The Ada LRM states that objects designated by access values "have no simple name;" that is, they cannot be referred to other than by way of the access type object.)

A solution is that an occurrence of $.(\#x) = t$ in the postcondition should be interpreted as a disjunction of conjunctions:

```
(#x = a1  and .a1 = t) or (#x = a2  and .a2 = t) or ... or .(#x)=t
```

where in all but the last disjunct the $a1$, $a2$, ..., range over all places of type value, and the last disjunct is the original predicate containing the term $.(\#x)$ itself. SDVS needs this disjunct for the cases where the new place has not existed previously, but nevertheless something can be proved about it based on other facts involving this new pounded place.[1]

An example of this latter phenomenon is given by

[1] Note that the continued occurrence of $.(\#x) = t$ in the postcondition does not make this a circular argument: we are merely proposing that the postcondition be expanded once in this manner.

```
[sd pre: (pcovering(all,x,.x),formula(inc5.sd))
    mod: (x,.x)
   post: (.(#x) = 1)]
```

where *inc5.sd* is

```
[sd pre: (true) mod: (x,.x) post: (.(#x) = 1)]
```

As an example where we need the disjunction, consider:

```
[sd pre: (pcovering(all,x,y,z),pcovering(all,x,.x),
   pcovering(all,y,.y),pcovering(all,z,.x),
   pcovering(all,z,.y),formula(inc3.sd),.(.x) = 1)
    mod: (z)
   post: (#z = 1, .(#x) = 1)]
```

where *inc3.sd* is

```
[sd pre: (true) mod: (z) post: (#z = 1)]
```

This is a true state delta; however, it just so happens that SDVS cannot prove the above state delta as written. It needs to be interpreted as if it were the following equivalent state delta:

```
[sd pre: (pcovering(all,x,y,z),pcovering(all,x,.x),
   pcovering(all,y,.y),pcovering(all,z,.x),
   pcovering(all,z,.y),formula(inc3.sd),.(.x) = 1)
     mod: (z)
    post: (#z = 1,
   (#x = .x & .(.x) = 1 or #x = .y & .(.y) = 1) or
     .(#x) = 1)]
```

SDVS proves this, requiring only the application of *inc3.sd*.

The following is *not* equivalent:

```
[sd pre: (pcovering(all,x,y,z),pcovering(all,x,.x),pcovering(all,y,.y),
   pcovering(all,z,.x),pcovering(all,z,.y),formula(inc3.sd),
   .(.x) = 1)
     mod: (z)
    post: (#z = 1,#x = .x --> .(.x) = 1,#x = .y --> .(.y) = 1)]
```

Notice that

$$[(p_1 \rightarrow q_1) \wedge (p_2 \rightarrow q_2)] \vee (\neg p_1 \wedge \neg p_2 \rightarrow q_3)$$

and

$$(p_1 \wedge q_1) \vee (p_2 \wedge q_2) \vee (\neg p_1 \wedge \neg p_2 \wedge q_3)$$

are not equivalent: let $q_1 = q_2 = T$ and $p_1 = p_2 = q_3 = F$.

As another example, note that the following state delta is not a theorem:

```
[SD pre: pcovering(all, x, y, z), pcovering(all, x, .x),
         pcovering(all, y, .y), .x = .y, formula(inc.sd), .(.x)=1, .z=0
  mod[]: z, .x, .y
   post: #z=1, #(#x)=1
```

where *inc.sd* is:

```
[sd pre: (true) mod: (z,.x,.y) post: (#z = 1)]
```

Nor is the following a theorem:

```
[sd pre: (pcovering(all,x,y,z),pcovering(all,x,.x),
         pcovering(all,y,.y),.x = .y,formula(inc1.sd),.(.x) = 1,
         .z = 0)
  mod: (z,x,y)
 post: (#z = 1,#(.x) = 1)]
```

where *inc1.sd* is:

```
[sd pre: (true) mod: (z,x,y) post: (#z = 1)]
```

because z and $.x$ are not necessarily disjoint.

Here is a correct version of the above example:

```
[SD pre: pcovering(all, x, y, z), pcovering(all, x, .x),
         pcovering(all, y, .y), pcovering(all, z, .x),
         pcovering(all, z, .y), formula(inc2.sd), .(.x)=1
  mod: z, x
 post: #z=1, #(.x)=1]
```

where *inc2.sd* is

```
[SD pre: (true)  mod; (z, x)  post: (#z=1)]
```

Another nontheorem is:

```
[SD pre:  pcovering(all, x, y, z), pcovering(all, x, .x),
          pcovering(all, y, .y), pcovering(all, z, .x),
          pcovering(all, z, .y), formula(inc2.sd), .(.x)=1
    mod:  z,x
   post:  #z=1, #(#x)=1]
```

Here is an example with $\#(.x)$: Consider the state delta *dots18.sd*:

```
   [SD pre: pcovering(all, x, y, z), pcovering(all, x, .x),
            pcovering(all, y, .y), pcovering(all, z, .x),
             pcovering(all, z, .y), formula(inc3.sd), .(.x)=1
       mod: z
      post: #z=1, #(.x)=1]
```

In words, this state delta formalizes the claim that given all the independence relations as specified in the "pcoverings", if the value of the contents of the place pointed to by x starts out at 1, and if z will become 1 in the future (per *inc3.sd*) with only z changing along the way, then z will become 1 at some future time when the value of the *new* contents of the *old* place pointed by x is still 1.

Here is the proof transcript as produced by SDVS 11:

```
<sdvs.1> prove
  state delta[]: dots18.sd
   proof[]:
 open -- [sd pre: (pcovering(all,x,y,z),pcovering(all,x,.x),
   pcovering(all,y,.y),pcovering(all,z,.x),
   pcovering(all,z,.y),formula(inc3.sd),.(.x) = 1)
     mod: (z)
    post: (#z = 1,#(.x) = 1)]

 Complete the proof.
<sdvs.1.1> goals

 g(1) #z = 1
 g(2) #x\136 = 1
<sdvs.1.1> apply
   sd/number[highest applicable/once]:

   apply -- [sd pre: (true)
        mod: (z)
       post: (#z = 1)]

 close -- 1 steps/applications
```

4 An Example Program

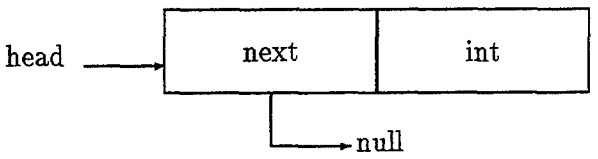
The procedure **LINKED** in Figure 1 on the next page accepts integer inputs (positive, negative, or zero), and when it receives a 0 input, it outputs the previous inputs in increasing order.

In this and the following two sections we study the program, first intutively, by showing on the next three pages successive states of the computation for the input stream 3, 7, 5, 0. Then we look at the operation of **LINKED** within the context of places of places, and finally in the last section we give some suggested translations into state deltas.

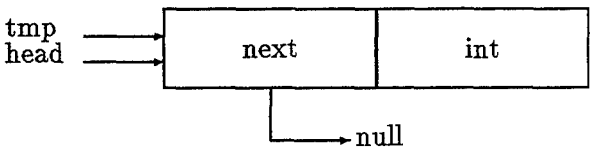
```
with TEXT_IO; use TEXT_IO;
with INTEGER_IO; use INTEGER_IO;
  procedure LINKED is
   type LINK;
   type P is access LINK;
   type LINK is
        record
             NEXT : P;
             INT  : INTEGER;

        end record;
   HEAD : P := new LINK; -- ''new'' means that .(.HEAD) ''exists''
   I    : INTEGER;

   procedure ADD_I_TO_LINKED_LIST is
     TMP : P := HEAD; -- Begin search of where to insert at start of list.
     begin
          while TMP /= null and then TMP.NEXT /= null
                    and then TMP.NEXT.INT < I loop -- TMP.NEXT.INT is initially 0
               TMP := TMP.NEXT;  -- so TMP = HEAD.NEXT
          end loop;
          TMP.NEXT := new LINK'(I, TMP.NEXT);-- Create new link and insert in list.
                                      -- TMP is still = HEAD.NEXT
    end ADD_I_TO_LINKED_LIST;

   procedure DISPLAY_LINKED_LIST is
     TMP : P := HEAD.NEXT;  -- Skip unused link at the head of the list.
      begin
           while TMP /= null loop
                 PUT(TMP.INT);       -- Print integer in the current link.
                 TMP := TMP.NEXT;    -- Go to next link in the list.
           end loop;
     end DISPLAY_LINKED_LIST;

    begin
         GET(I);
         while I /= 0 loop
                  ADD_I_TO_LINKED_LIST;
                  GET(I);
         end loop;
         DISPLAY_LINKED_LIST;
   end LINKED;
```

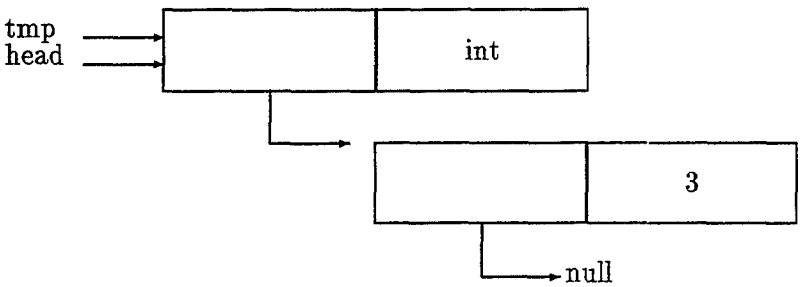
Figure 1: Procedure LINKED

Input: 3, 7, 5, 0

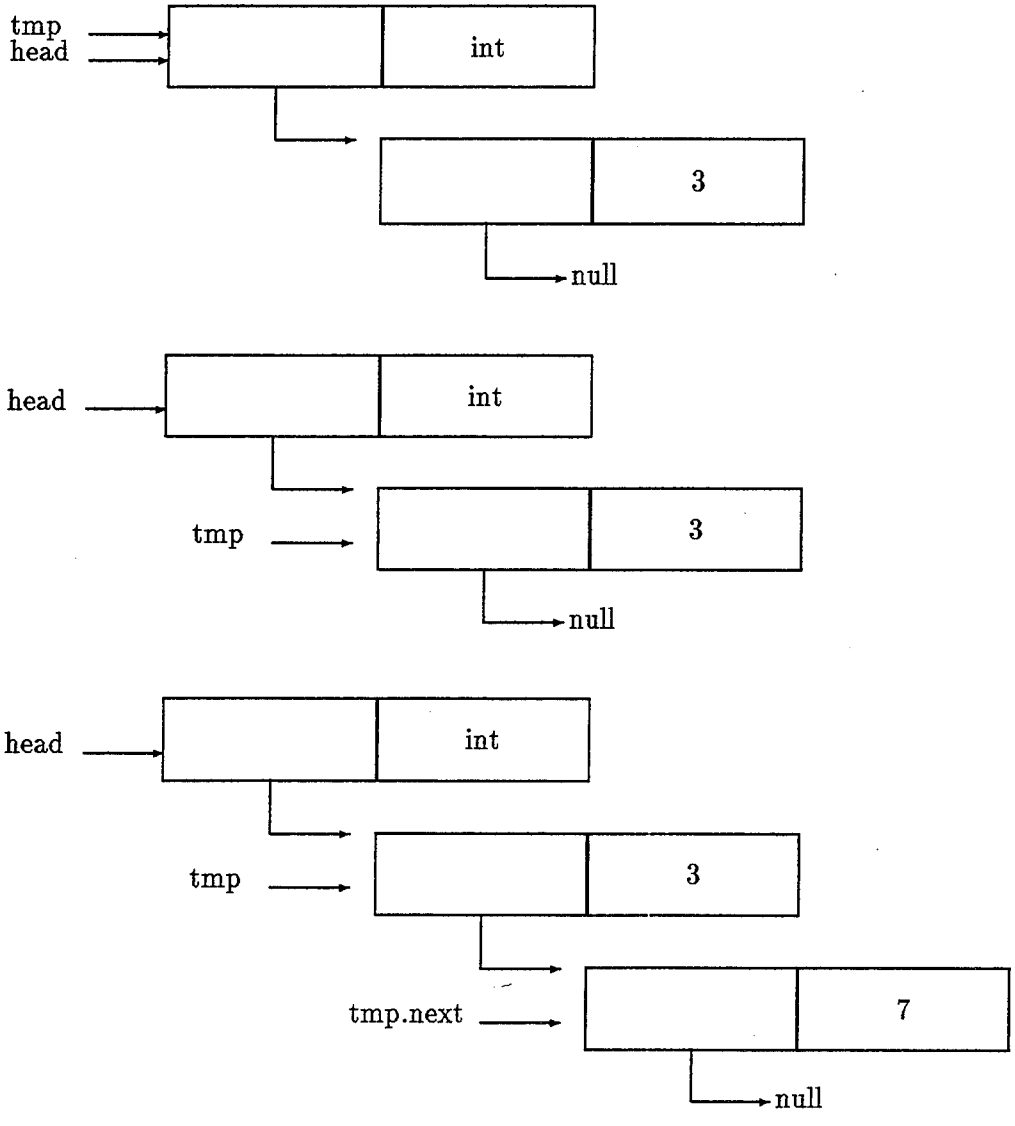


i = 3

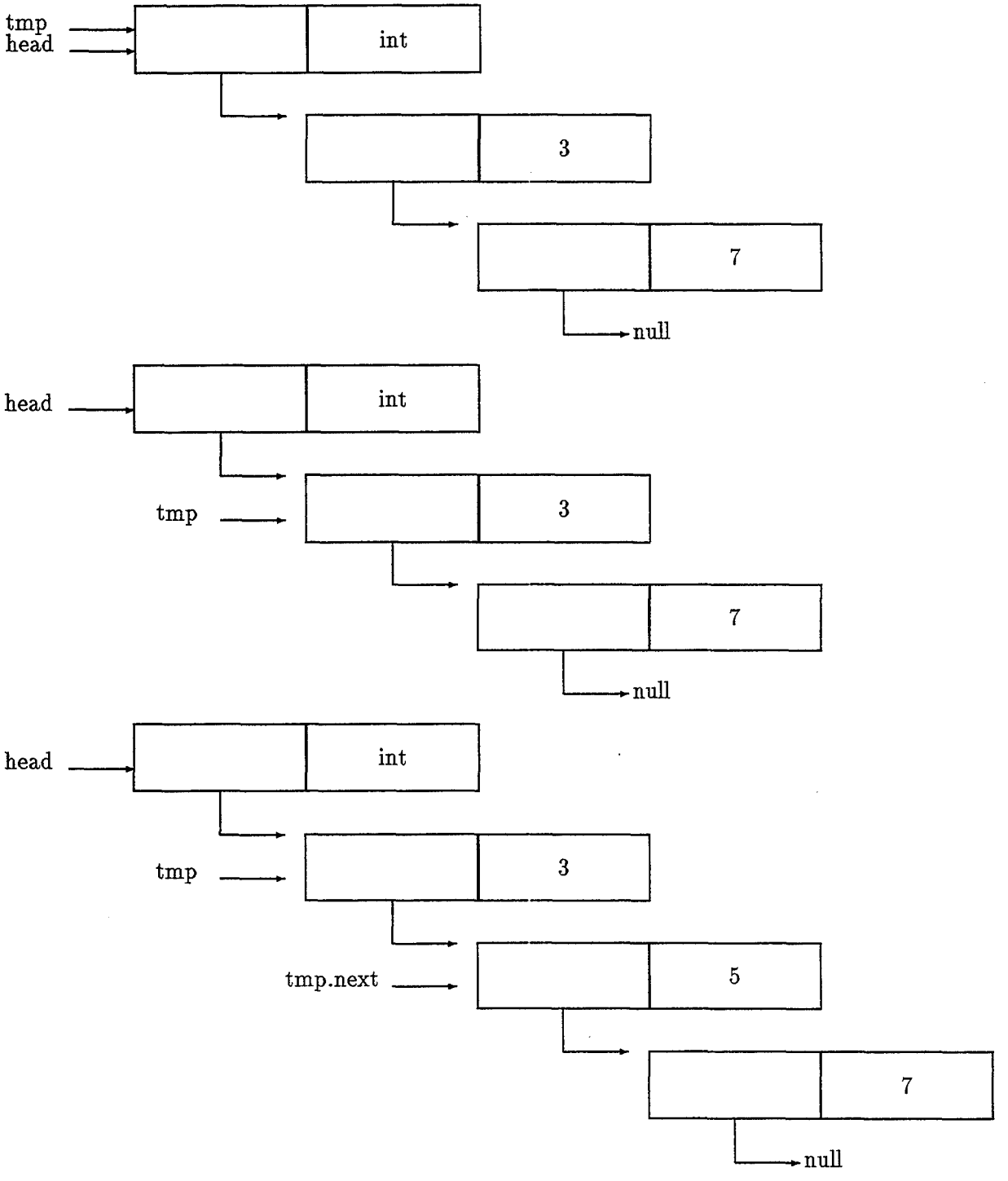




i = 7


tmp
head
int
3
null
head
int
tmp
3
null
head
int
tmp
3
tmp.next
7
null

i = 5


tmp
head
int
3
7
null
head
int
tmp
3
7
null
head
int
tmp
3
tmp.next
5
7
null

5 Derived Places of Places of Places

In this section we go into more detail on the treatment of arrays and records of places of places with the analysis of the program of the previous section as a motivating example.

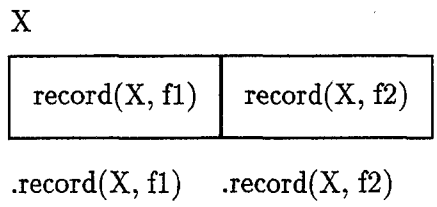
If X is a place of type array with elements of type integer (with indices of type integer, if there is a need to specify thus), then for each integer index i, $X[i]$ is a place with contents of type integer and $.X[i]$ is its value (an integer). In longer SDVS notation $X[i]$ is written as $element(X, i)$, and its contents as $.element(X, i)$.

Similarly, if X is a place of type record (with some specified fields f_i and records of specified types t_i), and f_i is a field name, then $record(X, f_i)$ is a place with contents of the apppropriate type (t_i) and $.record(X, f_i)$ is its contents.

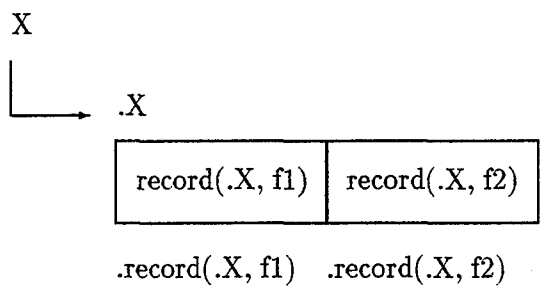
Now if X is a (second order) place of *access* type record and f is a field name, then its contents $.X$ is a place of type record. Then just as in the preceding paragraph, $record(.X, f_i)$ is a place and $.record(.X, f_i)$ is its contents.

Finally, consider a place X of access type record with a field of type access type. (If the field f_1 has the *same* access type as X, then it is a *circular* access type.) Again, $.X$ is a place of type record and $record(.X, f_i)$ is the place associated with field f_i. For f_1 this is an access type place, so $.record(.X, f_1)$ is again a place of the same type, and $record(.record(.X, f_1), f_1)$ is the access type place corresponding to f_1, and so on.

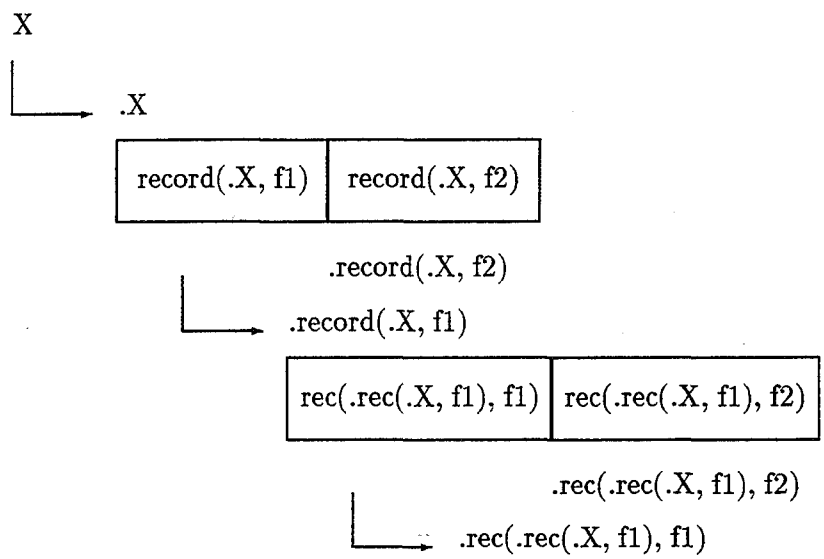
For X of type record:



For X of access type record:



For X of circular access type record:



6 Translating the Example Program

Below is the proposed SDVS translation of the above program, currently done "by hand."

First, in the declaration phase, the following are generated (as postconditions of successively nested state deltas; the types *incomplete* and *access* are proposed additions to SDVS):

```
 (alldisjoint(linked,.linked,link),
  covering(#linked,.linked,link),
  declare(link,type(incomplete)))

(alldisjoint(linked,.linked,p),
 covering(#linked,.linked,p)
 declare(p,type(access(link))))
```

Or perhaps simply,

```
(alldisjoint(linked,.linked,p),
 covering(#linked,.linked,p)
 declare(.p,type(link)))
```

To continue,

```
(alldisjoint(linked,.linked,link.int)
 covering(#linked,.linked,link.int),
 declare(link.int,type(integer)))

(alldisjoint(linked,.linked,link.next),
 covering(#linked,.linked,link.next),
 declare(link.next,type(p)))

(alldisjoint(linked,.linked,head),
 covering(#linked,.linked,head),
 declare(head,type(p)))

(alldisjoint(linked,.linked,i),
 covering(#linked,.linked,i),
 declare(i,type(integer)))
```

Now, in the procedure `ADD_I_TO_LINKED_LIST` we get

```
[sd pre: true
  comod: all
    mod: tmp
   post: declare(tmp, type(p)), #tmp = .head]
```

The tests `TMP /= NULL` and `TMP.NEXT /= NULL` are represented by `.TMP ~= NULL` and `.record(.TMP, NEXT) ~= NULL`, respectively. The test `TMP.NEXT.INT < I` is represented by `.record(.record(.TMP, NEXT), INT) lt .I`.

The assignment `TMP := TMP.NEXT` is represented by

```
[sd pre: true
  comod: all
    mod: TMP
   post: #TMP = .record(.TMP, NEXT)]
```

The assignment `TMP.NEXT := new LINK'(I, TMP.NEXT)` is represented by (something equivalent to)

```
[sd pre: true
  comod: all
    mod: record(.record(.TMP,NEXT),INT),
         record(.record(.TMP,NEXT),NEXT))
   post: #record(.record(.TMP, NEXT), INT) = .I,
         #record(.record(.TMP, NEXT), NEXT) = .record(.TMP, NEXT)]
```

The state delta representing the claim to be proven is (could be)

```
[sd pre: (ada(linked),
          (forall i (0 le i and i lt n --> .stdin[i] neq 0)))
  comod: (all)
    mod: (all)
   post: ((forall i (0 le i and i lt n-1 -->
                  #stdout[i] le #stdout[i+1])),
          (forall i (exists j (0 le i and i le n-1 -->
                  (0 le j and j le n-1 and #stdout[j]=.stdin[i])))))]
```

7 Conclusions

We have proposed a method for handling pointers and Ada access types in SDVS, i.e., a method that allows SDVS to translate and reason about Ada programs containing access types. The method is built upon "higher-order places", i.e., "places" that have other places as contents. SDVS 11 already handles correctly most of the basic operations of places of places.

We have given the state delta semantics for various Ada constructs involving access types, discussed the theory and implementation of higher-order places, and given a partially worked example of an Ada program involving access types.

References

[1] Grebyn Corporation, *The Annotated Ada Reference Manual (ANSI/MIL-STD-1815A-1983)*, June 1989.

[2] L. Marcus, "State Deltas with Places of Places." Verification Note 25, February 27, 1992.

[3] D. J. Naiditch, *Rendezvous with Ada*, (New York: John Wiley and Sons, 1989).

[4] A. D. McGettrick, *Program Verification using Ada*, (Cambridge, UK.: Cambridge University Press, 1982).

[5] D. L. Bryan and G. O. Mendal, *Exploring Ada, Volume 1*, (Englewood Cliffs, New Jersey: Prentice Hall, 1990).

[6] D. Guaspari, C. Marceau, and W. Polak, "Formal Verification of Ada Programs," *IEEE Transactions Software Engineering*, Vol. SE-16, pp. 1058–1075, September 1990.

[7] D. Guaspari, "Penelope for Ada," Email communication, April 1992.

[8] S. Meldal, "An Abstract Axiomatization of Pointer Types," in *Proceedings of the Symposium on Logic in Computer Science*, pp. 252–259, IEEE, 1989.

[9] H.-K. Hung and J. I. Zucker, "Semantics of Pointers, Referencing, and Dereferencing with Intensional Logic," in *Proceedings of the Symposium on Logic in Computer Science*, IEEE, 1991.

[10] V. Swarup and U. S. Reddy, "A Logical View of Assignments," in *Proceedings of the Conference on Constructivity in Computer Science*, June 1991.

[11] D. C. Luckham and N. Suzuki, "Verification of Array, Record, and Pointer Operations in Pascal," *ACM Transactions Programming Languages and Systems*, Vol. 1, pp. 226–244, 1979.

[12] N. Suzuki, "Analysis of pointer rotation," in *Proceedings 7th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 1–11, ACM, January 1980.

[13] W. E. Weihl, "Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables," in *Proceedings 7th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 83–94, ACM, January 1980.

[14] W. Landi and B. Ryder, "Pointer-induced Aliasing: A Problem Taxonomy," in *Proceedings 18th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 93–103, ACM, January 1991.

[15] W. Landi and B. Ryder, "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing," in *Proceedings ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248, ACM, June 1992.

[16] L. J. Hendren, J. Hummel, and A. Nicolau, "Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs," in *Proceedings ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 249–260, ACM, June 1992.

[17] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence Analysis for Pointer Variables," in *Proceedings ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 28–40, ACM, June 1989.

[18] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of Pointers and Structures," in *Proceedings ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 296–310, ACM, June 1990.